Investigating Patterns for the Process-Oriented Modelling and Simulation of Space in Complex Systems

Paul S. Andrews[1], Adam T. Sampson[3],
John Markus Bjørndalen[4], Susan Stepney[1], Jon Timmis[1,2], Douglas N. Warren[3] and Peter H. Welch[3]

[1]Department of Computer Science, University of York, UK, YO10 5DD
[2]Department of Electronics, University of York, UK, YO10 5DD
[3]Computing Laboratory, University of Kent, Canterbury, UK, CT2 7NF
[4]Department of Computer Science, University of Tromsø, Norway
psa@cs.york.ac.uk, A.T.Sampson@kent.ac.uk

Abstract

Complex systems modelling and simulation is becoming increasingly important to numerous disciplines. The CoSMoS project aims to produce a unified infrastructure for modelling and simulating all sorts of complex systems, making use of design patterns and the process-oriented programming model. We provide a description of CoSMoS and present a case study into the modelling of space in complex systems. We describe how two models – absolute geometric space and relational network space – can be captured using process-oriented techniques, and how our models can be refactored to allow efficient, distributed simulation. We identify a number of design, implementation and refactoring patterns that can be applied to future complex systems modelling problems.

Introduction

Complex systems consist of populations of low-level simple agents that interact concurrently with each other and their environment to exhibit high-level emergent behaviours. The modelling and simulation of complex systems is becoming increasingly important in a number of scientific disciplines. Real-world experimentation is often expensive and time-consuming; accurate simulations provide a powerful tool for understanding complex systems, and their results can help to direct future experimental work. There is therefore significant interest in the development of more effective tools and methodologies for modelling and simulation.

Under the banner of CoSMoS[1] (Complex Systems Modelling and Simulation infrastructure), we aim to develop a modelling and simulation infrastructure to allow complex systems to be designed, analysed and explored within a uniform framework. When completed, the CoSMoS system will allow users, guided by our methodology, to design, develop and analyse their own complex systems.

Our modelling process aims to be applicable to *generic* complex systems, and will make use of *patterns* and *refactorings*. Our simulation environment will be *massively-concurrent* and *distributed* through the use of the process-oriented programming model. This is important as our final infrastructure will be supported on a number of processing platforms including FPGAs, general-purpose PCs and clusters. We are adopting a case-study-based approach, modelling and simulating many complex systems to identify the necessary generic components. As we develop the case studies, we are consciously documenting and analysing how we are developing the models and simulations to extract the CoSMoS process. Through each case study this process is refined and augmented as new situations arise.

A number of tools for complex systems modelling and simulation already exist; for example, environments such as Breve[2] and Repast[3] allow for the development of agent-based complex systems simulations. The use of design patterns to document reusable solutions to complex systems modelling problems has previously been advocated by Wiles et al. (2005). CoSMoS differs in that it will bring together the modelling, simulation and analysis of generic complex systems under a single unified framework. Additionally, the massively-concurrent simulation environment will enable us to get closer to the scale of real-world complex systems.

In the context of CoSMoS, this paper presents a rationale for our approach and describes some initial steps towards achieving our aims. We start by describing why a process-oriented approach is applicable to complex systems, followed by some of the techniques we are employing in the pursuit of engineering reusable elements. We then present an investigation into space representations in various complex systems, and show how space can be modelled and simulated using a process-oriented approach. Finally we look at what our case study has shown us in relation to the aims of CoSMoS by identifying the kinds of patterns and refactorings that might be applicable to general complex systems.

A Process-Oriented Approach

In the process-oriented programming model, concurrent *processes* interact using mechanisms such as *channels* and

[1]http://www.cosmos-research.org

[2]http://www.spiderland.org

[3]http://repast.sourceforge.net

barriers. Process-oriented programming has a formal basis in process algebras such as CSP (Hoare, 1985) and the π-calculus (Milner, 1999). As a result, the semantics of communication and process composition are well-defined, and the behaviour of process-oriented programs can be reasoned about in a structured way. See Welch et al. (2006) for one example of this.

As the real world consists of concurrent, interacting entities, communicating process calculi – and therefore process-oriented programs – provide a natural way to construct models of the real world: entities map directly to processes, and the interactions between them are modelled by communications. There has been much research into modelling complex systems using process calculi such as the π-calculus (Phillips and Cardelli, 2007); this suggests that process-oriented programming techniques can be profitably applied to complex systems modelling and simulation.

Concurrent programs are well-placed to take advantage of multicore processors and multiprocessor hosts. The same programming models can be used to construct distributed systems; a process-oriented program can usually be refactored into a form that runs efficiently on a cluster.

Concurrency is traditionally seen as hard for programmers to get right, but this need not be the case. In a process-oriented system, a compiler can guarantee that processes are isolated from each other, and must communicate explicitly rather than sharing data. Processes may communicate references to data, but sending a reference to another process causes it to be lost from the sender (*data mobility*). These constraints combine to prevent common concurrency problems such as aliasing errors and race hazards. Other concurrency problems such as deadlock can be dealt with using simple design rules, proved correct by reference to the underlying process calculi.

One such set of design rules is the *client-server* pattern, in which client processes are connected to server processes by two-way bundles of channels. After the client initiates a communication, an arbitrary two-way conversation can take place between the client and server. Processes may act as both client and server, but they may be a client to only one server at a time. If there are no cycles in the directed graph of client-server relationships between a network of processes, the network will be both deadlock- and livelock-free (Martin and Welch, 1997). Many common patterns of concurrency – such as pipelines – already obey the client-server rules, but the rules also allow far more complicated process networks to be constructed safely.

Initial work on CoSMoS has used the occam-π process-oriented programming language[4]. In an occam-π implementation, process overheads are typically very small; a commodity PC can support millions of concurrent processes. Process creation and deletion is cheap, and communication is very efficient. This allows the programmer to take advantage of concurrency to simplify their program without worrying about adversely affecting performance. occam-π provides *channel bundles* as a language binding for client-server relationships; the endpoints of channel bundles can be communicated around at runtime (*channel mobility*), allowing dynamically constructed and reconfigurable networks.

[4] http://occam-pi.org/

Engineering Reusable Complex Systems

To develop an engineering approach to the modelling and simulation of generic complex systems, we must focus on *reusable* problem-solving techniques. Reusable techniques reduce the amount of work required in the development of a complex system, and lessen the risk of mistakes during specification or implementation. Additionally, because our systems will be built using common building blocks, it will be easier to combine models and simulations to study interactions between complex systems.

Our main tool for achieving reusability is one of the most successful and popular approaches in software engineering: patterns. The original idea of patterns comes from architecture courtesy of Alexander et al. (1977), who describe a pattern as "a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice". This idea was applied to object-oriented software design by Gamma et al. (1995), who identify four essential elements of a pattern:

Name: a brief phrase to summarise the pattern, and that can be used as part of a *pattern language* when discussing problems.

Problem: the situation in which the pattern may be applied.

Solution: the elements involved when solving the problem, and a guide to their implementation.

Consequences: the advantages and disadvantages of applying the pattern, allowing the designer to make a decision about the appropriateness of the pattern in their particular situation.

Most existing uses of patterns in software engineering use the object-oriented programming model, but patterns can be applied equally effectively to process-oriented programming and other models.

Although the original use of patterns in software engineering was at the design stage (hence "design patterns"), patterns have been developed for all stages of the software development process, from low-level coding right up to the design of development processes themselves. For example, antipatterns (Brown et al., 1998) can be used to document common mistakes and how they can be avoided and rectified. Other patterns include analysis patterns (Fowler, 1997),

coding patterns (Beck, 1997), and metapatterns that describe patterns themselves. Our modelling process aims to take advantage of patterns wherever possible, and in particular to develop pattern languages for: abstract computational representations of complex systems models; analysis of collective and emergent properties; and validity argument structures.

We are not alone in wanting to apply patterns in the field of complex systems. Wiles et al. (2005) suggest that attention to software engineering practice can benefit both modellers and biologists. We are in complete agreement with their assertion that the field of *in silico* modelling is reaching a point where common practices should be identified and formalised into patterns.

Refactoring is a particularly interesting concept from a patterns perspective. Refactoring is improving the structure of a model or program without changing its external behaviour. For our purposes, such refactorings might include improving the clarity of the model, or adapting simulations to take advantage of FPGAs or clusters. These approaches will be codified as refactoring patterns.

Space: a Case Study

As noted in the introduction, we have employed a case-study-based methodology in our investigation of the issues surrounding the modelling and simulation of complex systems. For the purposes of discussion in this section, we define a model to be an abstract logical representation of something we wish to better understand. A (computer) simulation is the execution of such a model over time, allowing us to analyse the model's behaviour.

The case study we describe here deals with the representation of space in a variety of different "textbook" complex systems. By "textbook", we mean well-understood examples of complex systems from the literature, such as boids (Reynolds, 1987), artificial ant behaviour (Amos and Don, 2007), L-systems (Prusinkiewicz and Lindenmayer, 1990) and scale-free networks (Barabasi and Bonabeau, 2003). We have chosen to study space initially because we feel that it is a universal property of complex systems models – and one that can be expressed in a wide variety of ways. (Future case studies will cover time and other commonalities.)

For millennia, philosophers have pondered the nature of space in the real world. At the turn of the eighteenth century, two very different views on space were held by Newton and Leibniz. Newton believed that space was *absolute* – independent of the objects that could exist within it. Leibniz defined space as *relational* – only existing in the relationships among the objects it contains (Garber, 1995; Giavotto and Michel, 2002).

In complex systems modelling, space is commonly represented using either of the Newton and Leibniz approaches. For example, simulations such as boids and artificial ants use an absolute geometric space, with a fixed area of space being defined in which all the agents reside. L-systems and scale-free networks, on the other hand, use a relative model of space: the model's idea of space comes solely from the relationships between L-system symbols or network nodes, and the geometry and size of space can change over time. Hybrid models of space can be built in which absolute geometric space is modelled by a sparse network of regions, created only when an agent or a behaviour requires their presence (Sampson et al., 2005).

The meaning of points in a model's representation of space may correspond to locations in physical space (with one, two or three dimensions), or to something more abstract. For example, a point in absolute "shape-space" (Perelson and Oster, 1979) could represent a set of parameters describing the shape of a molecule in an immunological model (Hart and Ross, 2004).

Space models such as absolute geometric space may be continuous, or quantised into a grid of discrete positions. Some models may be built on either space representation; for example, interactions between cells in the bloodstream may be modelled using distance calculations in continuous space, or by looking at neighbouring cells in discrete space. The choice of space representation – and, if a discrete model is used, the fineness of the grid – will often affect the dynamics of the model.

Previous Work

The TUNA project[5] was the feasibility study that led to CoSMoS, investigating tools for engineering emergent behaviour in nanite systems. The primary case study was the simulation of artificial blood platelets, which would staunch wounds in a blood vessel as an emergent behaviour. A number of different models were considered.

Initial efforts focused on cellular automata in one, two and three dimensions. The first models used simple, deterministic rules, and were built using the CSP process calculus and analysed using FDR and Probe[6]. Later models were extended to include platelet activation and diffusion of chemical factors, and were implemented using occam-π. The completed blood clotting simulation (Ritson and Welch, 2007) runs in three dimensions, using VTK[7] for volumetric visualisation, and allows the user to interactively create wounds in the blood vessel. The program can be distributed across a cluster of commodity PCs, enabling simulations with tens of millions of agents to run at acceptable speeds. The resulting simulation demonstrates several behaviours seen in real-world haemostasis, and can be used to perform simple *in silico* experiments.

Design patterns were developed for the efficient simulation of grid-based space using process-oriented techniques, in which space cells are represented as processes (Sampson et al., 2005). The construction of space processes can

[5] http://www.cs.york.ac.uk/nature/tuna/

[6] http://www.fsel.com/software.html

[7] http://www.vtk.org/

be delayed until agents move into them, allowing a sparse, lazy representation of space. Agents can "sleep" by not engaging in synchronisation when their state is unlikely to change, saving processor time. Concurrent access to shared resources can be managed safely using barrier synchronisation and *phases* (Barnes et al., 2005). Agents can migrate transparently between different hosts in a cluster.

Modelling Continuous Space

The TUNA project examined grid-based models of space. These are easy to reason about, but they are insufficiently accurate for the purposes of many interesting models. Continuous space is generally more useful when modelling real-world systems. In continuous space, it immediately becomes harder for agents to find nearby agents with which to interact; they cannot simply look in the neighbouring locations, but must consider the distances between them. In a trivial implementation of continuous space, all agents have knowledge of all other agents, but this is inefficient (as well as a poor model of the real world); we need a representation of the world with an idea of locality. We would also like to be able to take advantage of the space-modelling efficiency patterns that were developed for TUNA.

As a first case study, we implemented the simulated bird flocking model *boids* (Reynolds, 1987). At each time step, boids adjust their velocity based on the following rules:

Collision Avoidance: avoid collisions with nearby objects

Velocity Matching: try to match velocity with nearby boids

Flock Centring: try to stay close to nearby boids

This results in the emergent flocking behaviour. We used a two-dimensional model of space, although boids (and our space model) would work equally well in three dimensions.

The approach we took was to divide the space up into regions, with each region represented by a *location* process. Each location contains an arbitrary number of *agent* processes (boids and obstacles), and keeps track of a local position for each, relative to the centre of the region. Locations are connected much as cells are in a grid-based model; each location process has a shared channel bundle which its neighbours have access to, and provides a server interface that allows clients to enter a cell, move around, and retrieve a list of agents along with their positions.

The first thing that each boid must do on each timestep is to "look around" for other agents in its neighbourhood. To do this, the boid needs to gather the contents of all the cells that intersect with the region it can see. We have restricted our agents to seeing a circular region with a diameter of at most one location, which means that it is sufficient to look at the location the agent is in and the eight surrounding locations. Figure 1 shows a boid's field of vision in the partitioned continuous space model.

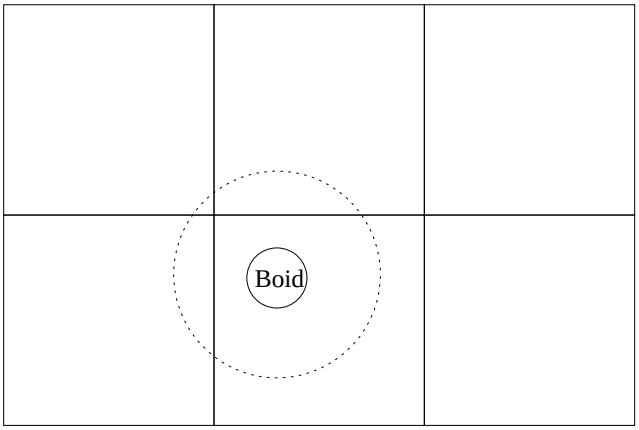

Figure 1: A boid's field of vision (dotted line) in the partitioned continuous space

Since all agents in each location need to look at the same set of nine locations, we can save some effort by delegating this task to a shared *viewer* process. Each location has a viewer process permanently attached to it, and on each time step the viewer updates its view of the surrounding world. The viewer process then provides a server interface to the agents in the corresponding cell which allows the agents to obtain their local view.

In order to guarantee that the agents see a consistent view of the world, we must make sure that all the viewers are updated after the agents have finished moving, but before they look again at the start of the next time step. We therefore divide each timestep into multiple phases, with a global barrier synchronisation between each phase:

- In phase 1, the viewers request the contents of the surrounding cells.
- In phase 2, the agents request their view from the viewers, compute their new velocity, and send movement messages to their locations.

Once a boid has looked around, it decides in which direction to move by sending a movement vector to its location. The location responds by updating the boid's position. If the boid remains within the same location, no further action is necessary. However, if the boid has moved outside the bounds of the location, it must be moved into the next location in the correct direction. This is achieved by the response to a boid's movement request being "you must move into this location". This approach makes it possible to move across multiple locations in one movement step: upon entry, the first new location can respond immediately with another "you must move into this location" message, thus the agent reaches the correct target location by an iterative process.

In order to avoid complicating every agent with code to handle movement, we inserted an additional *agent manager*

process that hides the details of this from the agent itself. The manager provides a simplified interface to the agent, supporting only "move" and "look" requests. Adding this level of indirection simplified later work: it made it possible to have arbitrary behaviour inside the space model that is not visible to the agent itself.

Distributed Continuous Space

We used *pony* (Schweigler, 2006) to distribute the boids simulation across a cluster of hosts, with each host simulating a rectangular region of space modelled by several location processes. *pony* provides networked channel bundles for occam-π programs, with exactly the same semantics as local channel bundles; the only visible difference is the significantly increased latency compared to local communications. To obtain good performance, it is generally best to engineer a distributed application in such a way as to minimise the number of cross-host communications, and perform cross-host communications in parallel as far as possible.

To start with, we just modified the existing simulation to set up the same network of processes across a distributed application. The resulting simulation worked exactly as before, but ran very slowly; furthermore, it got even slower as boids migrated between hosts. There were two major sources of inefficiency:

- Neighbouring viewer processes must request the same view information from a location on the other side of a network link. For local communications this is not a problem, since only a reference is transferred; for network communications the data must be copied.

- More seriously, agent processes continue to run on the host they were started on, so once moved to a new host, every communication they do is across a network link.

To solve the viewer problem, we applied the *remote proxy* distributed computing pattern (Roth, 2002) in the form of *ghost* processes, which cache the contents of a location on the other side of a network link. Viewer and agent processes that would ordinarily communicate with a remote location are instead given a channel bundle to the corresponding local ghost (which provides the same server interface as the remote location). Since ghost processes must update their cached contents before viewer processes try to read it, we needed to introduce an additional phase to the simulation:

- In phase 1, the ghosts request the contents of their corresponding locations.

- In phase 2, the viewers request the contents of the surrounding cells.

- In phase 3, the agents request their view from the viewers and send movement messages to their locations.

To solve the agent problem, we introduced the idea of *agent migration*. In response to moving to a location on a different host, an agent can be told to suspend itself: pack up its internal state and terminate on the originating host. The state is moved to the destination host, where a new agent process is started using the existing state. This is straightforward to implement: when an agent attempts to move into a ghost (rather than a real location), the ghost replies to the agent with a "suspend" message, and then signals the real target location to spawn a new process.

A sample process network at a host boundary in the final model is shown in Figure 2. The cycle time of the resulting simulation is approximately equal to that of the single-host simulation plus the network latency. We ran the simulation across a cluster of networked PCs: the cycle time remained approximately constant as the simulation was scaled from two to eight hosts. This is as expected, since each host only needs to communicate with its immediate neighbours.

In order to increase performance further, we have experimented with more efficient strategies for inter-host communication. Relaxing the normal CSP channel semantics for networked channels to permit *asynchronous* delivery of messages means that channel communications no longer need to be acknowledged by the receiving host, approximately halving the network latency and thus reducing the simulation cycle time. Since network channels are only used by the ghost processes, we can simply adjust the ghost protocol so that it still behaves correctly with asynchronous communication. In the future, we plan to experiment further with batching of messages in order to reduce TCP overheads and permit message compression.

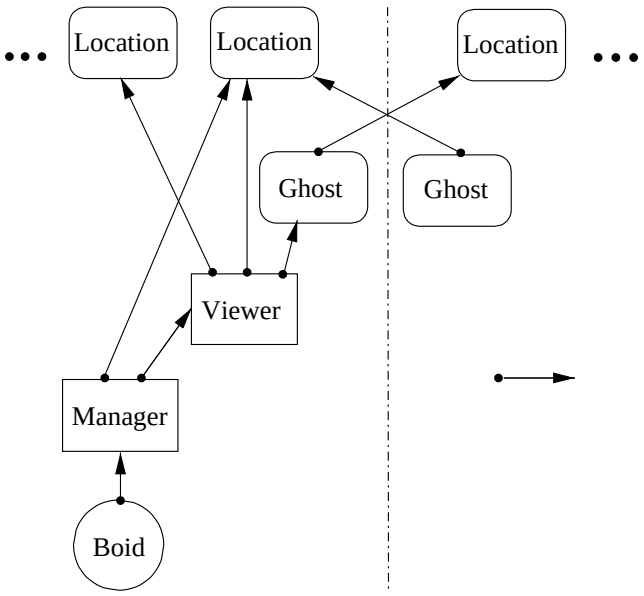

Figure 2: Process network at a host boundary in distributed boids

Different Model, Same Space

To demonstrate the reuse of our space model, we implemented a different complex system simulation on top of our continuous space model. The complex system we chose was ant-based annular sorting (Amos and Don, 2007), in which ants sort eggs into rings by size by picking up poorly-placed eggs and dropping them when they find a better location.

As with boids, we modelled ants and eggs as agent processes. To allow ants to carry eggs, we extended our system so that agents could pick up other agents (removing them from their locations), and put them down elsewhere. No further changes were necessary to the space model, suggesting that it had potential for reuse in other similar simulations.

Our model of continuous space is generic enough that it supports different kinds of agent with differing behaviours. They might sense different aspects of the environment and have different goals. The location and viewer processes report the locations of all nearby agents, whereas the agent manager and agent processes provide the specific agent behaviour. Additional sensory modes and/or noise could be added to the underlying location and viewer architecture, which agent processes can filter accordingly.

Constructing Network Space: Edges as Channels

Network space is an example of relational space in which the network nodes are the space-defining objects. Networks can be modelled very straightforwardly in a process-oriented way: nodes are processes, and edges are channels. As an example, we implemented L-systems (Prusinkiewicz and Lindenmayer, 1990): rewriting systems based on a formal grammar (a set of rules and symbols) that can be used to model growth processes such as plant development and organism morphology. For example, a very simple grammar might be defined as follows:

Symbols: $A, B, +, -$

Start symbol: A

Rules: $(A \rightarrow B - A), (B \rightarrow A + B)$

Here we have two symbols A and B which are transformed by the corresponding rules, and two symbols $+$ and $-$ which do not change. By specifying a start symbol, we can iteratively apply the L-system rules in parallel so that our symbol string grows with each iteration as follows:

Iteration 0: A

Iteration 1: $B - A$

Iteration 2: $A + B - B - A$

Iteration 3: $B - A + A + B - A + B - B - A$

L-systems are often visualised by translation into "turtle graphics" instructions. For example, if we define variables to mean "move straight ahead one unit", $+$ to mean "turn right $60°$" and $-$ to mean "turn left $60°$", we end up with the visualisation shown in Figure 3.

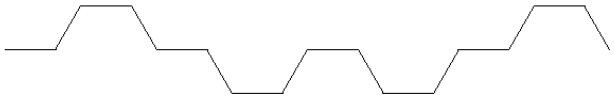

Figure 3: Example L-system after 4 iterations

We model an L-system as a process network in which the L-system symbols are represented as separate processes connected by channels. Here, the channels provide the ordering of the L-system string. At each iteration of the L-system, each process holding a variable symbol applies its corresponding transition rule, and replaces itself with the processes and channels corresponding to the expansion of the symbol.

Figure 4 shows a step-by-step application of the $B \rightarrow A + B$ transition rule. In the first step a new process is spawned that contains the last symbol in the transition rule. This process is then given the end of the right hand channel, and a new channel is created to connect this new process to the original B process that is being transformed. We work from the right-hand side of the rule so that the process network re-configures from the inside, with the right hand channel to the rest of the process network only having to be rewired once. After the first new process is connected, we work through the transition rule creating new processes and channels and reconfiguring existing channels where necessary. So, in our example, the next step involves inserting a process containing a $+$ symbol. As a last step, the original process that has undergone the transition rule has its symbol changed to the leftmost symbol in the rule – in this case, B is changed to an A. New processes are created on demand by a *factory* process.

The simulation is visualised on each iteration using a display process, which is connected to both ends of the chain of symbol processes, forming a ring. To visualise the network, the display process sends a channel end to the first symbol process, which outputs its symbol down the channel, then passes the channel end on to the next process. This repeats until the channel end makes it all the way around the ring and returns to the display process, which then knows it has gathered the complete state of the network, and draws it to the display using turtle graphics rules.

Constructing Network Space: Edges as Processes

A scale-free network is one in which some nodes are highly connected, whilst most have few connections. There is no notion of a typical node in the network: its properties are independent of the number of nodes. Examples include

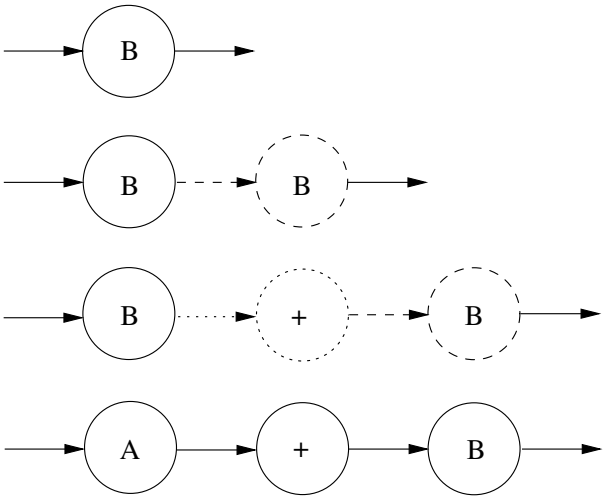

Figure 4: Network reconfiguration during a rule application

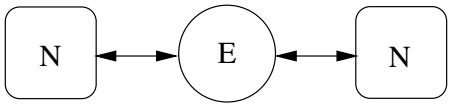

Figure 5: An implementation of an undirected edge, where N denotes a node process and E an edge process

the World-Wide Web: nodes are pages, and edges are hyperlinks; research collaborations: nodes are scientists, and edges are co-authorships; and protein regulatory networks: nodes are proteins, and edges are interactions amongst proteins (Barabasi and Bonabeau, 2003). They are scale-free because their properties are similar regardless of how many nodes are present in the network; for example, the distribution of path lengths between pairs of arbitrary nodes will not change as more nodes are added. Barabasi et al. (2000) have shown that by utilising a scheme known as preferential attachment – in which nodes prefer to connect to other nodes that are already well-connected – you can grow a generic network that is scale-free.

In the L-systems example above, edges were represented as channels. Channels in occam-π are directed, and are used directly in the L-systems network to match the left-to-right ordering of symbols. In a scale-free network, edges may be undirecte with no natural ordering . We modelled this by representing edges as processes with separate channels connecting them to the nodes on each side; an example is shown in Figure 5. This is a more flexible model, since there is no need for an explicit ordering, and edges may have their own behaviours if necessary. For example, if an edge needs to be reconnected between a different pair of nodes, it can take part in the decision and reconnection process itself.

To implement a growing scale-free network with preferential attachment, we start by creating two node processes and linking them by an edge processes. Next a controller process iteratively forks a new node process and connects it to a pre-defined number of new edge processes. For each new edge process, a randomly selected pre-existing node is selected and connected to the edge process. This random selection is biased towards highly connected nodes, thus implementing preferential attachment.

In the same way that we can apply a continuous space model to both boids and ant-based annular sorting, we can easily adapt the scale-free network with preferential attachment model to implement a small-world network (Watts and Strogatz, 1998) instead. This reuses the same node and edge processes, but changes the way they are connected together.

Space: the Results

Our space models produced several useful, reusable components. Both space models were successfully applied to more than one complex system example with minimal work. In addition, we have identified a number of initial design patterns, which we can can categorise into four groups: *modelling*, *implementation*, *optimisation* and *refactoring* patterns. Patterns we identified included:

Distributed Continuous Space (*modelling*): by dividing continuous space into regions, we can efficiently implement local vision in a distributed simulation.

Agent Process (*modelling*): agents are modelled as concurrent processes that interact within a space. The space may be modelled explicitly using additional processes, or may be implicit in the relationships between the agents.

Factory Process (*implementation*): factory processes spawn new processes at runtime in response to requests from other processes. They provide a common *context* for the newly-created processes, and hide details of creating, configuring and connecting up new processes behind an interface. (This is the process-oriented equivalent of the *abstract factory* pattern (Gamma et al., 1995).)

Ghost Location (*optimisation*): when refactoring a simulation to run in a distributed manner, a ghost process can cache the contents of a remote location to avoid repeated network communication. (This is an application of the existing *remote proxy* pattern.)

Agent Migration (*optimisation*): in a distributed simulation, an agent can be suspended and moved to a different host, in order to minimise the number of network communications it must do.

Reification (*refactoring*): creating a process (a "thing") to represent a relationship between two other processes. For example, a directed link between two processes can simply be a channel, but an undirected or buffered link can be better modelled as a process.

In addition, we found possible patterns related to visualisation and the modelling of time, which we are investigating.

Conclusion

In this paper we have outlined CoSMoS, a planned modelling and simulation infrastructure for the investigation of generic complex systems. We are using the process-oriented programming model, owing to the natural analogies between processes and complex system agents and the ability to construct massively-concurrent and distributed simulations. CoSMoS will promote reusable modelling techniques through the development of pattern languages.

We have studied the modelling and simulation of space in complex systems in the context of reusable modelling techniques. We have shown how two very different spaces, a geometric continuous space and a arbitrary network space, can be modelled and simulated in a reusable way, and have identified a number of design and refactoring patterns.

The next step for CoSMoS will be to start modelling and simulating some more detailed complex systems based on real-world observations and data. This will help identify further generic complex system components, and aid the development and validation of our method and toolset.

Acknowledgements

This work is part of the CoSMoS project, funded by EPSRC grant EP/E053505/1 and a Microsoft Research Europe PhD studentship.

References

Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I., and Angel, S. (1977). *A Pattern Language*. Oxford University Press.

Amos, M. and Don, O. (2007). An ant-based algorithm for annular sorting. In *Proceedings of the 2007 IEEE Congress on Evolutionary Computation (CEC)*, pages 142–148. IEEE Press.

Barabasi, A., Albert, R., and Jeong, H. (2000). Scale-free characteristics of random networks: the topology of the world-wide web. *Physica A*, 281:69–77.

Barabasi, A. and Bonabeau, E. (2003). Scale-free networks. *Scientific American*, 288:60–69.

Barnes, F. R. M., Welch, P. H., and Sampson, A. T. (2005). Barrier synchronisation for occam-pi. In *2005 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, pages 173–179. CSREA Press.

Beck, K. (1997). *Smalltalk Best Practice Patterns*. Prentice Hall.

Brown, W. J., Malveau, R. C., McCormick III, H. W., and Mowbray, T. J. (1998). *AntiPatterns: refactoring software, architectures, and projects in crisis*. Wiley.

Fowler, M. (1997). *Analysis Patterns: reusable object models*. Addison Wesley.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Garber, D. (1995). Leibniz: Physics and philosophy. In Jolley, D., editor, *The Cambridge Companion to Leibniz*. Cambridge University Press.

Giavotto, J. and Michel, O. (2002). Data structures as topological spaces. In *3rd International Conference on Unconventional Models of Computation (UMC02)*, pages 137–150. Springer.

Hart, E. and Ross, P. (2004). Studies on the implications of shape-space models for idiotypic networks. In *3rd international Conference on Artificial Immune Systems (ICARIS04)*, pages 413–426. Springer.

Hoare, C. A. R. (1985). *Communicating Sequential Processes*. Prentice Hall.

Martin, J. M. R. and Welch, P. H. (1997). A Design Strategy for Deadlock-Free Concurrent Systems. *Transputer Communications*, 3(4).

Milner, R. (1999). *Communicating and Mobile Systems: The π-calculus*. Cambridge University Press.

Perelson, A. S. and Oster, G. F. (1979). Theoretical studies of clonal selection: Minimal antibody repertoire size and reliability of self–non-self discrimination. *Journal of Theoretical Biology*, 81(4):645–670.

Phillips, A. and Cardelli, L. (2007). Efficient, correct simulation of biological processes in the stochastic pi-calculus. In *Computational Methods in Systems Biology (CMSB07)*, pages 184–199. Springer.

Prusinkiewicz, P. and Lindenmayer, A. (1990). *The Algorithmic Beauty of Plants*. Springer-Verlag.

Reynolds, C. W. (1987). Flocks, herds, and schools: A distributed behavioral model. In *14th Annual Conference on Computer Graphics and Interactive Technologies (SIGGRAPH87)*, pages 25–34. ACM.

Ritson, C. G. and Welch, P. H. (2007). A process-oriented architecture for complex system modelling. In *Communicating Process Architectures 2007*, pages 249–266. IOS Press.

Roth, J. (2002). Patterns of mobile interaction. *Personal Ubiquitous Computing*, 6(4):282–289.

Sampson, A. T., Welch, P. H., and Barnes, F. R. M. (2005). Lazy Cellular Automata with Communicating Processes. In *Communicating Process Architectures 2005*, pages 165–175. IOS Press.

Schweigler, M. (2006). *A Unified Model for Inter- and Intra-processor Concurrency*. PhD thesis, Computing Laboratory, University of Kent, Canterbury, UK.

Watts, D. J. and Strogatz, S. H. (1998). Collective dynamics of 'small-world' networks. *Nature*, 393:440–442.

Welch, P. H., Barnes, F. R. M., and Polack, F. A. C. (2006). Communicating complex systems. In *11th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS06)*. IEEE Press.

Wiles, J., Geard, N., Watson, J., Willadsen, K., Mattick, J., Bradlet, D., and Hallinan, J. (2005). There's more to a model than code: understanding and formalizing *in silico* modeling experience. In *2005 Workshops on Genetic and Evolutionary Computation (GECCO)*, pages 281–288. ACM.